DEFENCE RESEARCH AGENCY MALVERN

MEMORANDUM No. 4587

Z THROUGH PICTURES

Author: G P Randell

DEFENCE RESEARCH AGENCY,
MALVERN
WORCS.

MEMORANDUM No. 4587

DEFENCE RESEARCH AGENCY

Memorandum 4587

Title:	Z Through Pictures
Author:	G P Randell
Date:	March 1992

Abstract

The purpose of this memorandum is to propose diagrams which may be generated from a Z specification either fully automatically or with human intervention. The main purpose of such diagrams is to illustrate the specification and thus help a reader to understand it and to validate it. Examples of the diagrams proposed are given.

Contents

1. Introduction

The need for validating a system specification is paramount. That is, a specification of a system, whether it be software, hardware or a combination of both, must be examined to ensure that it accurately reflects the real requirement, before the development of the system begins. Validation is often thought of as answering the question: "Are we building the right system?". Various techniques may be used for validation, from reviews and walk-throughs to rapid prototyping and analysis. Whichever techniques are used, one in particular is almost certain to feature, and that is reading the specification. In order to be sure that a specification does reflect the real needs, it must be understood.

A specification must not only be valid (that is, describe the right system), it should also be complete, consistent, and unambiguous. Often specifications are written in a natural language such as English. But English is often ambiguous, and it is very difficult to show convincingly that such specifications are complete or consistent. In an attempt to improve the quality of specifications, formal languages have been developed. These languages are characterized by being based on sound underlying mathematical principles, such as set theory. They are precise and unambiguous, having a formal syntax and semantics, and allow for the possibility of proving that designs and the final implementation meet the specification.

However, formal specifications can be difficult to read as they use mathematical symbols which may appear daunting at first sight. They also contain more detail than is often the case in a natural language specification, as the use of mathematics forces precision. Formal languages require mathematical training to use well. Thus validating a formal specification can be difficult.

The purpose of this memorandum is to describe techniques for producing diagrams which display the structure of a formal specification and help the reader understand it. Diagrams are easier to take in, and can show a variety of detail. One of the main barriers to understanding formal specifications is that they are often presented bottom-up. That is, basic type information and operations are first specified, and these are built up into more complicated operations and functions. This is especially the case with specifications written in the formal language Z [1,2], with which this paper is primarily concerned. However, it is often easier to understand a system top-down. That is, first concentrating on sub-systems, then sub-sub-systems, and so on, to the required level of detail. Thus diagrams which show the structure of a formal specification, how all the different pieces fit together, would be very useful and would provide much needed context.

Apart from structure, formal specifications can also be interpreted in terms of data flows. Rules for translating a suitable Z specification into a data flow diagram have been developed [3,4], and these may be used to produce a hierarchy of diagrams, corresponding to the structure of the specification.

In this memorandum, a set of possible diagrams is presented, and illustrated by example. Throughout, it is assumed that only those diagrams appropriate will be produced in any one case. That is, a "tool-kit" approach is adopted, whereby a selection of techniques is made available, but the use of these techniques is not prescribed.

Diagrams are not only useful for the reader of a specification, but are also useful for the writer too. In particular, drawing diagrams may make mistakes or bad structure apparent. It is often too easy to get involved in the detail of a formal specification, while losing track of what areas of the system being specified have been covered. It is also easy to lose track when combining small operations into larger ones, particularly remembering to specify what happens to all state variables, and all outputs, as noted in [5].

Diagrams may also help with maintenance of the specification. That is, if a diagram shows which definitions are dependent on which, then the impact of change can more easily be assessed and those components of the specification which are affected can be traced. Without such a diagram, this becomes very difficult.

For the purpose of this memorandum, it is assumed that the specification has been written in Z and, moreover, that it has been structured. This may either be done, if the Z support tool allows, by splitting the specification into modules or documents, or simply by dividing the specification document itself into chapters or sections. The reader is assumed to have some familiarity with Z to understand some of the more detailed diagrams. It is the intention of this work to allow a reviewer of a specification to more easily understand the specification, and to see how it fits together.

The remainder of this memorandum is structured as follows. Section 2 proposes a set of diagrams which have been called "maps". These diagrams are to show which things are used where, and to make the structure of the specification more apparent. They provide a top-down view of the bottom-up specification. Section 3 briefly covers the production of data flow diagrams. Section 4 proposes other diagrams, which do not come into either of these first two categories, and section 5 contains the conclusions of the memorandum.

Annex A contains an example Z specification, which will be used as the basis for the examples given in the main body of the memorandum.

2. Maps

The first category of diagram proposed is that of maps. These diagrams are to show the structure of a specification, how things fit together. Maps may be produced at many levels of detail, showing just the schemas defined, or how types are constructed, or showing where all the components in a schema are defined.

As with all diagrams, these maps should be kept to an understandable size. A large complicated diagram hinders more than it helps. A hierarchy or set of diagrams should be produced rather than one large one. Maps should also be used in conjunction with an index, which enables the reader not only to find the name of a component schema, say, but also to find exactly where in the specification document that schema is defined. The production of such an index is outside the scope of this memorandum.

Four sorts of maps are proposed: module charts, which show how the different sections of the specification fit together; schema usage maps, which show how state and operation schemas are built up and used; data maps, which show how complicated data structures are built up; and detailed schema maps, which show how all the components of a schema are constructed. These are discussed in more detail below.

The following simple specification of a library system will be used in this section and throughout the remainder of this memorandum as the basis for all the example diagrams. The complete specification is given at Annex A and is structured into five parts:

1. The specification of the types of data which are used by the different sub-systems of the library.

2. The book-buying sub-system which is responsible for ordering new books from a bookshop, paying for them when they arrive, and preparing them for being put on the library shelves ready to be loaned to borrowers.

3. The borrowers sub-system which is responsible for keeping track of registered borrowers and the books they have out on loan, including functions to add new borrowers, for a borrower to take out and return a book, and to lose a book.

4. The shelf-stackers sub-system, which is responsible for stacking returned and new books on the shelves, and for sending damaged books away to the bookbinders for repair and getting them back again.

5. The overall system, where the operations which cross the sub-system boundaries are specified. This specification actually only contains one such operation.

2.1 Module Charts

Considering first the overall structure of a specification, a map based on the module chart used in the Yourdon method [6] may be drawn to show how the different sections of that specification interact with one another. Such a diagram would present a top-down view of the system being specified, and allow the reader of the specification to understand how the sections fit in to the overall system description.

Figure 1 below shows a simple module chart for the example specification.

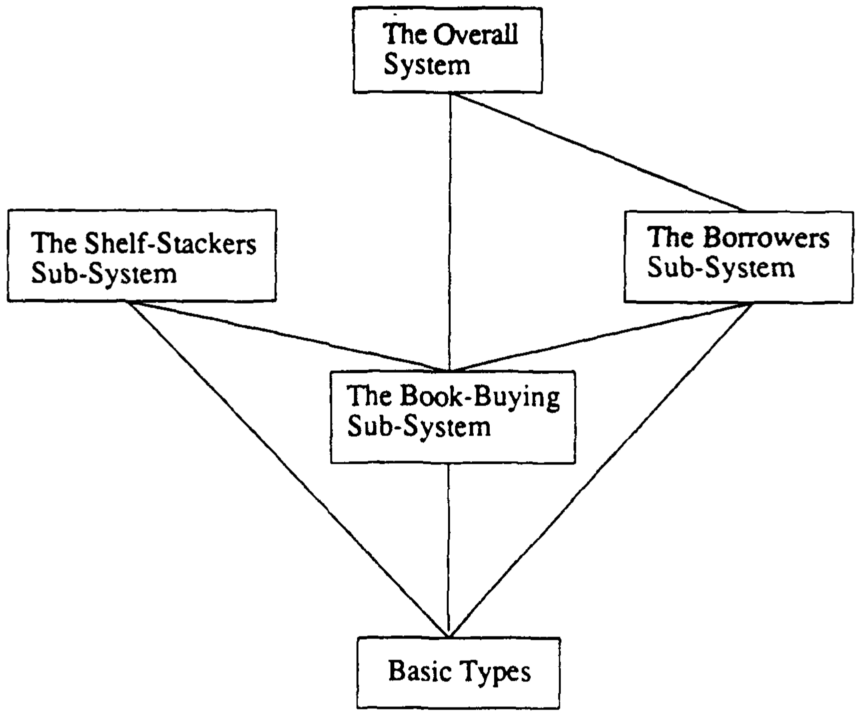

Figure 1 - A Simple Module Chart

This diagram shows the five parts of the specification, in a top-down manner. That is, the overall system specification at the top relies on definitions in the book-buying sub-system and in the borrowers sub-system. Similarly, the shelf-stackers sub-system use definitions from the basic types part of the specification, and so on.

However, a simple chart such as that shown in Figure 1 above gives no indication of which components from any section of specification are used by any other section. A logical extension of this sort of map is to label the lines joining any two sections of the specification in the simple chart with the names of the components, be they types, constants, schemas or whatever, being used. This will result in an annotated module chart.

For the simple example, this would produce the chart shown in Figure 2 below.

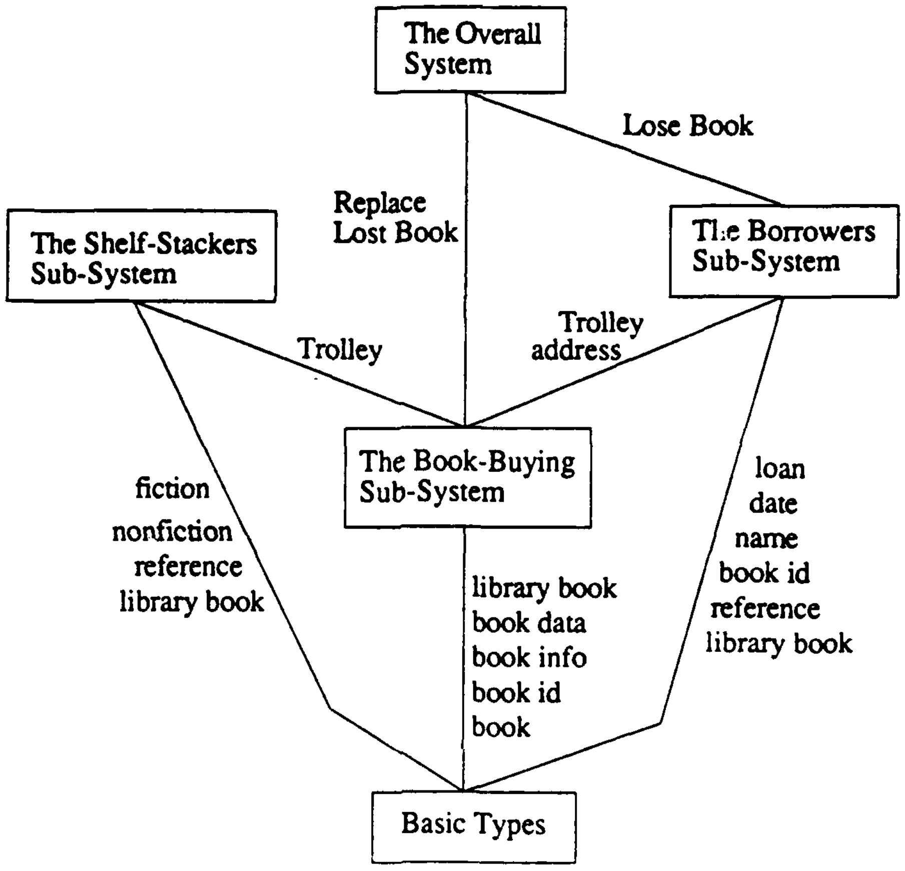

Figure 2 - An Annotated Module Chart

This more complicated diagram shows the identifiers used by the different parts of the specification. Thus the borrowers sub-system makes use of the definitions of "loan", "date", "name", "book id", "reference" and "library book" from the basic types section, together with the definition of "address" and "Trolley" from the book-buying sub-system. From this chart, it can be seen that "Trolley" is declared in the book-buying sub-system, and used by both the other sub-systems. So perhaps it would have been better to put this declaration in the basic types section.

The usage of different components by different sections of specification is shown up by the annotated module chart, rather more than it is in the specification itself. Thus these diagrams give a visual way of checking the coupling between different sections of specification, that is, how much one section relies on the definitions in another. The more reliance there is between sections the more difficulties may arise in the future when the specification is changed or refined towards an implementation. For example, changing the structure of a type used by many sections will require all those sections to be checked and possibly changed as well.

Thus these diagrams are not only useful for providing a top-down view of a bottom-up specification, but are also useful for maintenance purposes.

2.2 Schema Usage Maps

Just as module charts show the overall structure of a specification at the module level, schema usage maps can show the connections between schemas, that is, which schemas are included within which others. This sort of map is particularly useful to show how schemas describing operations on state are built up. As before, these maps are top-down, so the schema at the top includes those schemas below it to which it is linked.

The maps can be used to show not just the break-down of one operation schema into its component parts, but to show how all the schemas in one section of specification are related.

An example schema usage map for the book-buying sub-system of the library example is shown in Figure 3 below.

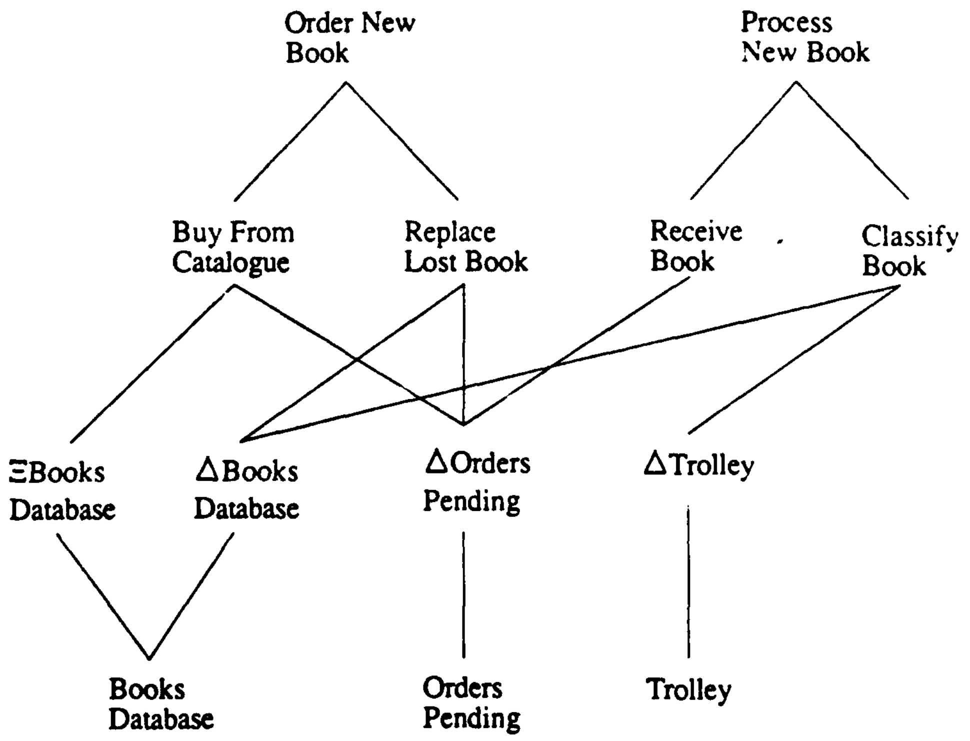

Figure 3 - A Map of the Book-Buying Sub-System

This diagram shows the names of all the schemas used in the book-buying sub-system. This map is top-down, so, for example, the schema

"Replace Lost Book" is built up from the schemas "ΔBooksDatabase" and "ΔOrdersPending". The schema is actually defined as:

```
┌ ReplaceLostBook ──────────────────────────────────────┐
│ ΔBooksDatabase
│ ΔOrdersPending
│ lost? : book_id
│ order! : order_form
├───────────────────────────────────────────────────────
│ order!.o_n = new_number
│ order!.b_i = (bookmap lost?).b_i
│ order!.add = our_address
│ ordermap' = ordermap ∪ {new_number ↦ order!}
│ bookmap' = { lost? } ⩤ bookmap
│ where
│ new_number : order_number | new_number ∉ dom order_map
└───────────────────────────────────────────────────────┘
```

This schema has two components other than schemas. If all of the components of this schema were to be shown on a diagram then the idea of a map would need to be extended to show how types are constructed, as well as schemas. Diagrams describing the construction of types are discussed in sections 2.3 (Data Maps) and 4.4 (Structure Charts) below. A composite map showing all the details of a schema's signature, including both schema inclusions and declarations, is proposed in section 2.4 below.

Figure 3 may be simplified by making use of the delta and xi conventions of Z. Given a schema S then ΔS and ΞS are usually defined as:

```
┌ ΔS ───┐        and     ┌ ΞS ────┐
│ S                      │ S
│ S'                     │ S'
└───────┘                ├────────
                         │ θS = θS'
                         └────────┘
```

Provided delta and xi are used like this, the construction of ΔS and ΞS are known for any schema S and so this information is not really needed on the map. Thus either the bottom line or the one above it of Figure 3 could be removed.

These schema usage maps are just as useful as module charts: they show how the schemas in a section of specification fit together. They are also just as useful for maintenance purposes.

2.3 Data Maps

In this section a map very similar to those in the previous sections is proposed, to describe how complicated data structures are built up. These diagrams work in exactly the same way as the maps proposed so far, with complicated structures at the top of the diagram being broken down into basic types at the bottom. In Z, types are constructed from given sets and the type constructors product and power set. For example, a function is really a set of pairs of elements. Showing this level of construction is not very helpful. A much more useful diagram would show constructs at a "user level", such as functions, free types, or schemas, for example. The maps proposed in this section are at this level. An alternative diagram for data structures, the structure chart, is proposed in section 4.4.

The first example is a map showing how a free type is constructed. The Z definition, taken from the example specification at Annex A, is:

```
book_type ::= fiction | nonfiction | reference
```

The map produced is a top-down tree-shape diagram and is shown in Figure 4 below.

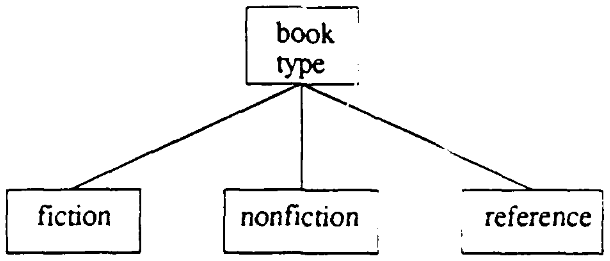

Figure 4 - The "book type" Free Type

A general free type may contain constructor functions mapping elements of complicated sets onto elements of the free type. In this case a tree structure may be drawn with the name of the constructor function labelling the appropriate line. To show this, consider the free type:

$$type1 ::= a \mid b \mid c \langle\langle C \rangle\rangle \mid d \langle\langle D \rangle\rangle$$

where C and D are any sets.

This may be represented by the diagram shown in Figure 5 below.

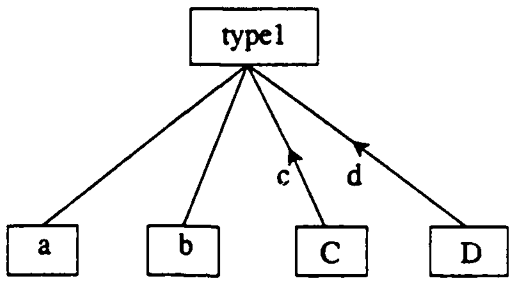

Figure 5 - Representing a General Free Type

This diagram has arrow heads on the lines which have associated constructor functions. These just show the direction of the function, from the set "C" to the free type "type1" in the case of the constructor function "c", for example.

The idea of using tree-shape diagrams to show elements of a free type was also used in [7].

Maps can also be drawn for a data structure defined as a schema, in much the same way. The name of the schema is at the top of the tree, and the components of the schema are linked to it in the now familiar way. For example, consider the schema:

```
┌ book_info ──────────────┐
│ ti  :  title
│ au  :  author
│ pub  :  publisher
│ p_d  :  publication_date
└─────────────────────────┘
```

where "author", "publication date", "publisher" and "title" are all given sets.

The corresponding data map is given in Figure 6 below.

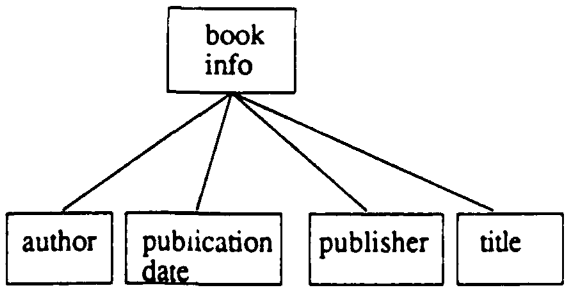

Figure 6 - The "book info" Schema Type

Note that the lowest level on any of these data maps consists of given sets or simple branches of free types (constants).

The problem with the maps shown in Figures 4 and 6 above is that it is not apparent from just the diagrams that one represents a free type and one represents a schema. Thus an annotation is proposed, to make this distinction clear. This is the reason for drawing boxes around the names of data: it means symbols can be added to the top right corner of the box, to indicate what sort of data structure is being defined. The symbols proposed are summarized in the following table:

Symbol	Meaning
[ ]	Given set
↔	Relation
→	Function[1]
⊏	Schema
ft	Free type[2]
ℙ	Power set
×	Product

[1] Different function arrows may be used to show total or partial injections, surjections and bijections.

[2] In addition, arrows will be used to show the direction of constructor functions, as in Figure 5 above.

Note that no symbol is given for constants in a free type definition. Using these symbols, the free type "book-type" and the schema "book-info" would be described by Figures 7 and 8 below, respectively.

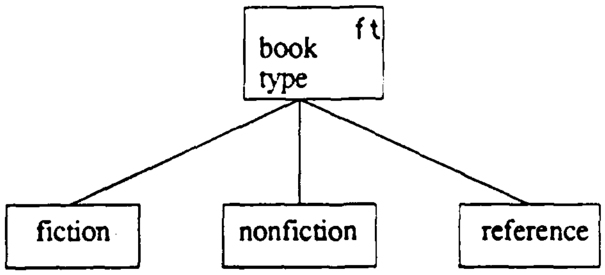

Figure 8 - The Annotated Data Map for "book-type"

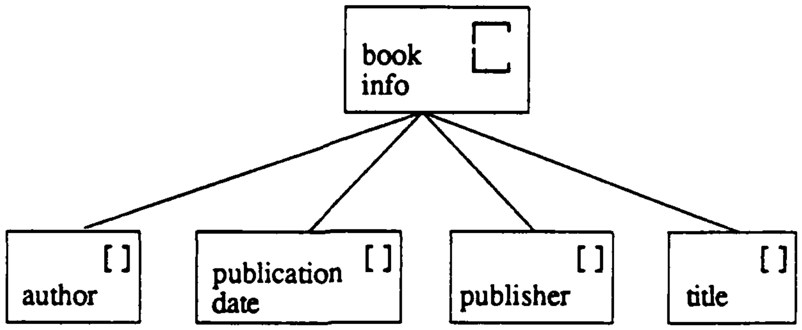

Figure 8 - The Annotated Data Map for "book-info"

More complicated data maps can be drawn, showing the relationships between all the data structure in a section of specification, if required. Care must however be taken to ensure that the diagram does not become too complicated, else it will become unreadable and thus defeat its purpose. It is better to draw several smaller diagrams than one too large one.

If a diagram is becoming too large, then a double box should be drawn around one or more items, and these expanded on separate diagrams. Figure 9 below shows two non-annotated diagrams linked in this way (although having two separate diagrams is not strictly necessary in this case).

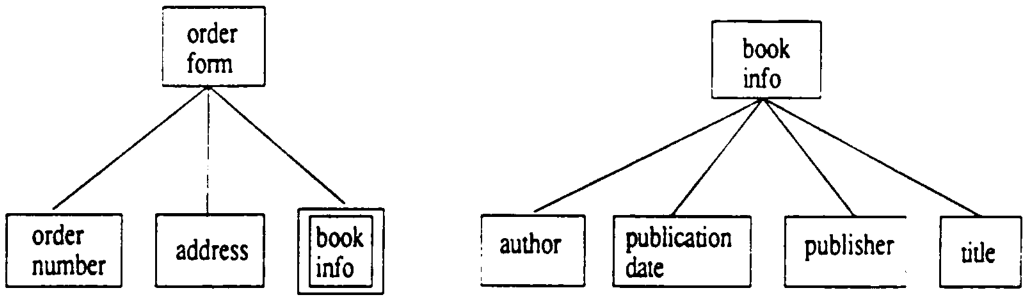

Figure 9 - Linked Diagrams

In the left hand diagram, a double box is drawn around "book info" to indicate there is another diagram (shown on the right) which explains how this is constructed.

As before, these maps are useful not only to show a top-down view and to explain where things are used, but they are also useful for subsequent refinement and maintenance when the impact of change needs to be assessed.

2.4 Detailed Schema Maps

The final sort of map proposed is one which combines the previous two. That is, it shows all the detail of the schema inclusions and the declarations in a schema's signature. A sensible size diagram should be produced, therefore it will often be unwise to show the complete break-down of the types.

For example, consider the schema describing the operation to replace a lost book, given in section 2.2 above. The detailed schema map for this is given in Figure 10 below.

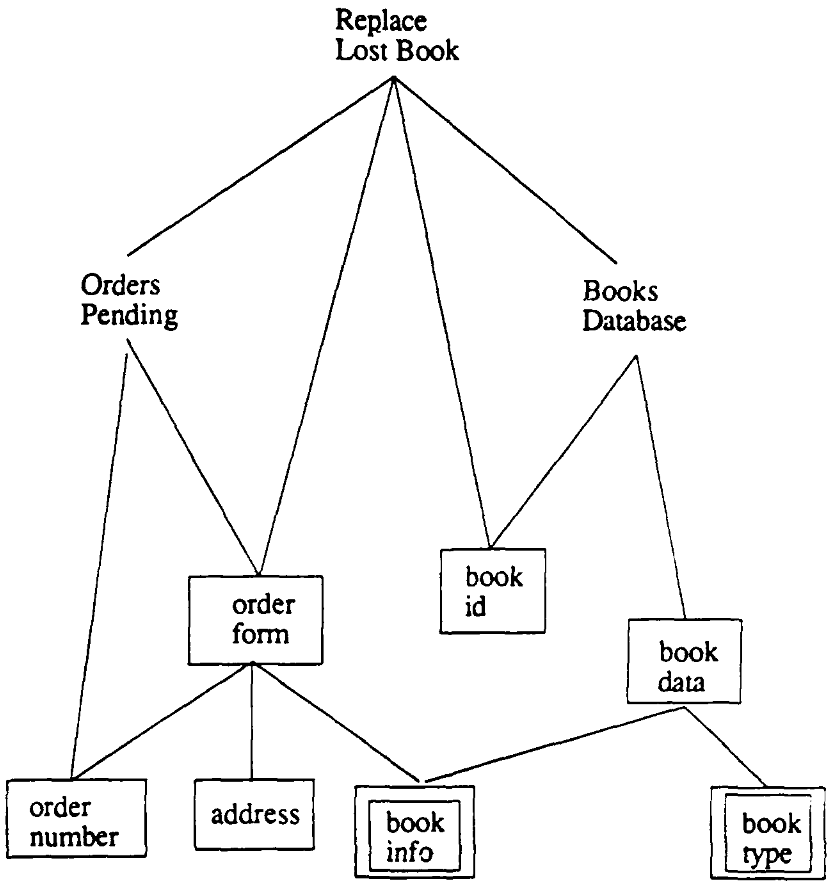

Figure 10 - The "Replace Lost Book" Operation

As in section 2.3 above, double boxes are drawn around data structures which are further described by their own data maps. The same annotations could be used to add detail to this map.

3. Data Flow Diagrams

Data flow diagrams (DFDs) are diagrams for concisely describing the flows of information to, from and within a system. As such they are ideal for understanding and validating the description of the functionality of a system. This is because:

- they are a pictorial representation which is relatively quick to produce and easy for users to understand,
- interconnections are shown graphically, making the effects of change easy to identify, and
- the DFD technique of top-down expansion, also called levelling, lets the reader progress from high to low levels of detail in a controlled way.

DFDs contain four types of symbols (elements). These are:

1. External entity - a source or recipient of data outside the system. It is usually what makes things happen inside the system.
2. Process - an activity which transforms or manipulates data.
3. Data store - a collection of any type of data in any form.
4. Data flow - shows a movement of data, with an arrowhead indicating the direction of flow.

The conventions of a data flow diagram are shown in Figure 11 below.

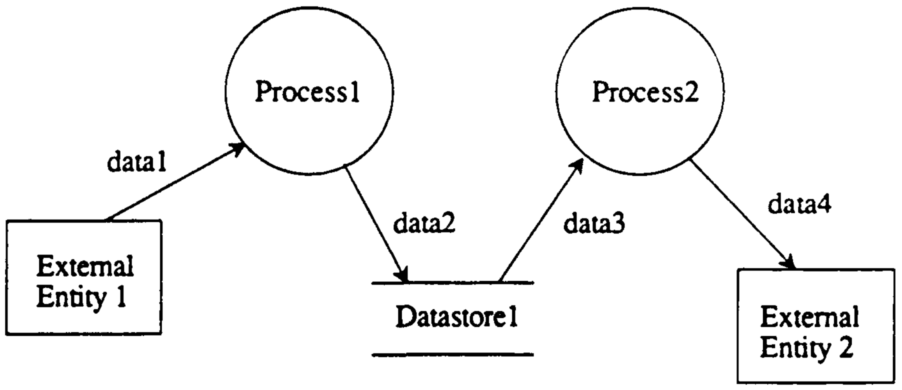

Figure 11 - A Data Flow Diagram

The conventions shown here are those according to the Yourdon method. In this method, processes are represented as circles (bubbles), data stores by open ended rectangles, and external entities as rectangles. Data flow diagrams are also used in SSADM (Structured

Systems Analysis and Design Method), but with slightly different conventions.

DFDs are developed as a hierarchy of diagrams. The top level diagram is called the Level-1 diagram and shows the basic characteristics of the system. The main system functions are represented by process bubbles on the Level-1 DFD and each of these bubbles may be expanded into a lower level DFD. Each process bubble can be thought of as a "window" into the diagram at the next level down in the hierarchy. The lower level diagram contains an expansion of the detail of the higher level.

A context diagram is also often drawn, which is a special type of DFD. It is the most abstract description of a system, and contains just one process bubble representing the whole system. The context diagram is used mainly for scoping the system as it shows those external entities with which the system interacts and the data which they give rise to or use.

Provided a Z specification is written in a state plus operations style, data flow diagrams are ideal diagrams to accompany that specification. Rules for generating a data flow diagram from a Z specification have been given elsewhere [3,4]. This memorandum gives examples of the diagrams generated from the specification at Annex A, and explains how the diagrams produced may be organised into a hierarchy for presentation purposes.

The hierarchy of DFDs for the example specification is shown in Figure 12 below.

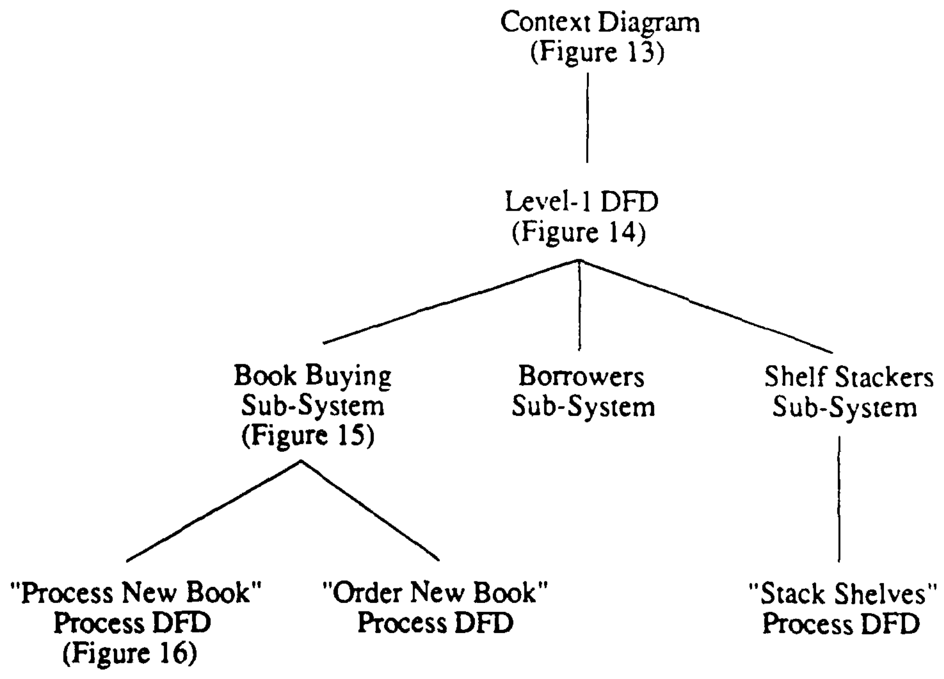
Context Diagram (Figure 13)
Level-1 DFD (Figure 14)
Book Buying Sub-System (Figure 15)
Borrowers Sub-System
Shelf Stackers Sub-System
"Process New Book" Process DFD (Figure 16)
"Order New Book" Process DFD
"Stack Shelves" Process DFD

Figure 12 - The Hierarchy of Data Flow Diagrams

The example specification is in five parts: one section introducing basic types, three sub-systems, and one part which describes the only operation which crosses sub-system boundaries. There is little point drawing a data flow diagram for the first section, as it contains no state schemas and no operation schemas.

The context diagram for this example is shown in Figure 13 below.

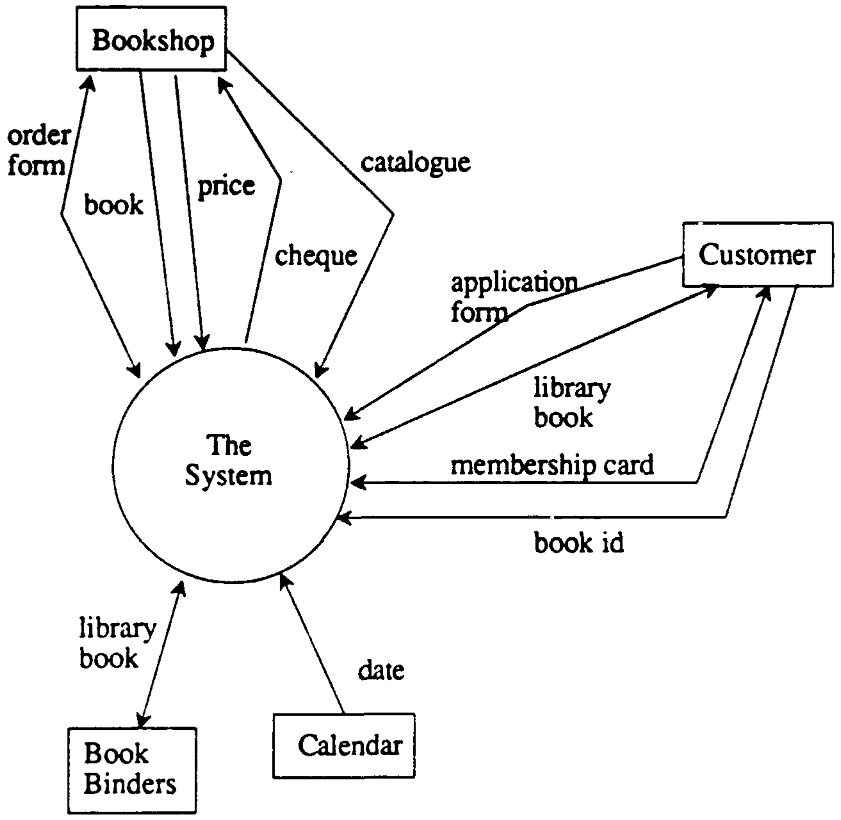

Figure 13 - The Library System Context Diagram

The context diagram is the most abstract data flow diagram of a system, and is used mainly to scope the system, that is, to show which external entities the system interacts with. For simplicity, if a data flow goes from an external entity to a process and back, then rather than draw two arrows one is drawn, but with arrowheads on both ends. For example, a membership card is given by a customer (borrower) to the library when a book is borrowed (part of the borrowers sub-system), and is returned on completion of the transaction, thus one double-headed arrow, labelled "membership card", has been drawn between the customer and the borrowers sub-system. This removes clutter and so aids readability of the diagram.

The next level down in the hierarchy is the Level-1 diagram, shown in Figure 14 below.

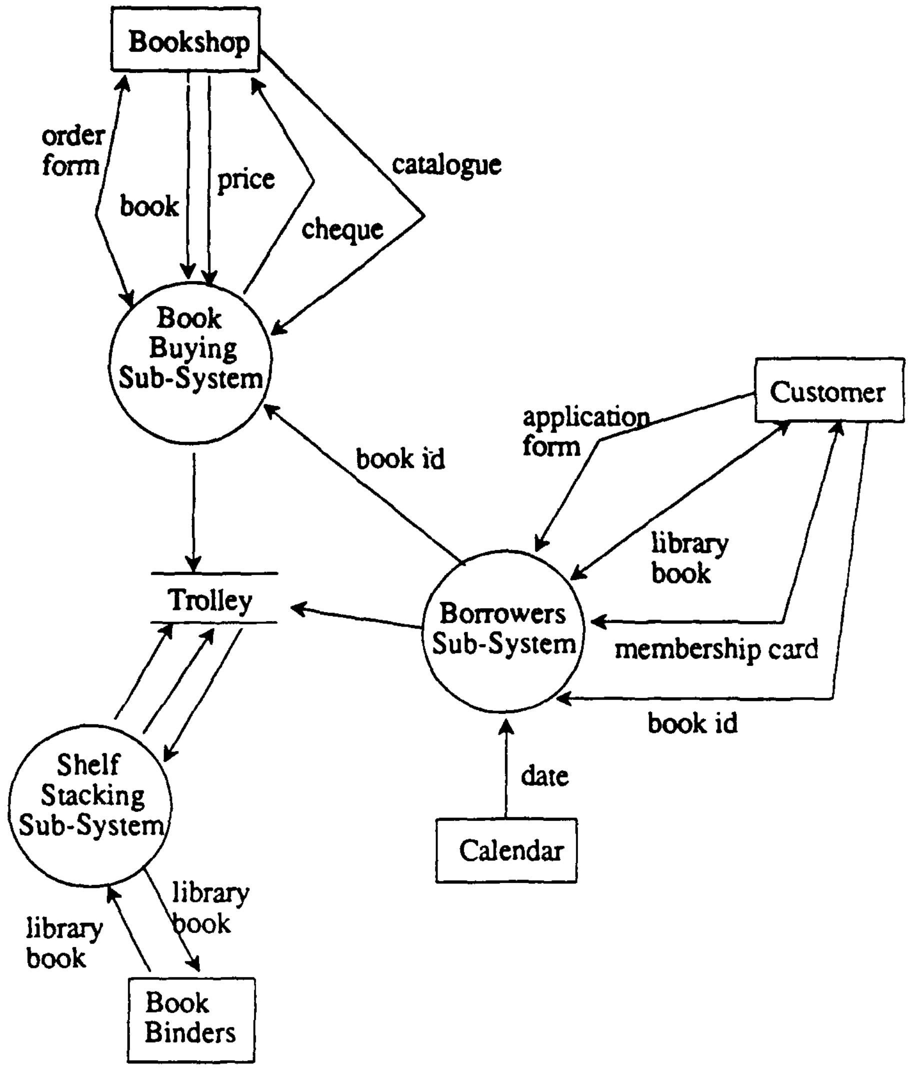

Figure 14 - The Level-1 DFD

This diagram has three process bubbles, corresponding to the three sub-systems. The data flow from the borrowers sub-system and the book-buying sub-system is the identifier of a lost book which is sent from the borrowers sub-system to the book-buying sub-system who will replace the book by buying a new copy. This flow is derived from the fifth part of the specification which specifies the "LostBook" operation which crosses these two sub-system boundaries. As before, double-headed arrows have been used to keep the diagram readable.

The trolley data store also appears at the top level, as it is used by more than one sub-system. Those data stores which are used only by one sub-system, such as the book-buying sub-system's books database, are within that sub-system and so do not appear on the top level data flow diagram.

Below the Level-1 diagram is one diagram for each sub-system. In order to generate a DFD describing a sub-system, the state schemas (which will be translated into data stores), and operations (which will be translated into processes) must be identified. In the book-buying sub-system there are three state schemas: "BooksDatabase", "OrdersPending" and "Trolley". Deciding which operations should go on the DFD for this sub-system is more interesting. This is because the operation "OrderNewBook" is defined in terms of "BuyFromCatalogue" and "ReplaceLostBook", and also "ProcessNewBook" is defined in terms of "ReceiveBook" and "ClassifyBook". The most sensible approach seems to be to put only the top-level processes on the DFD, namely "OrderNewBook" and "ProcessNewBook". This results in the data flow diagram shown in Figure 15 below.

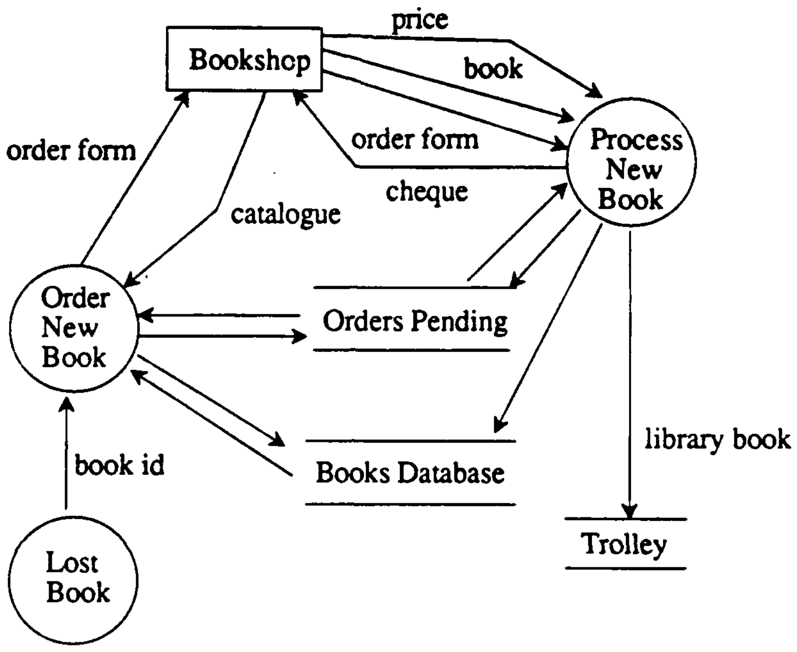

Figure 15 - The Book-Buying Sub-System DFD

The process "Lost Book" on this diagram is actually a process from the borrowers sub-system. It is the source of one of the inputs to the "OrderNewBook" process. The task of adding the data flows and external entities is not automatic, but must be done by hand. This may include adding elements from other sub-systems to complete the diagram. Some methods would leave the "Lost Book" process off this diagram, and have the "book id" data flow coming from nowhere. Similarly the "Trolley" database, which is actually used by all three sub-systems could be left off. Whether these should be added or not is a matter of preference.

The lowest level diagrams in the hierarchy describe the operations within the sub-systems which are built up from several schemas. In this example, the only lower level diagrams which could usefully be produced are for the "Order New Book" and "Process New Book" processes from the book-buying sub-system, and for the "Stack Shelves" process from the shelf-stackers sub-system. The diagram for the first of these is shown in Figure 16 below.

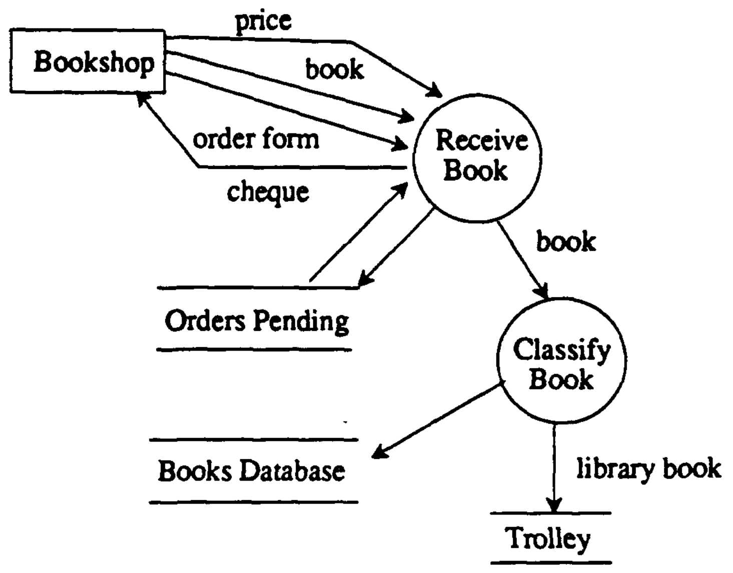

Figure 16 - The "Process New Book" DFD

This diagram has been completed, in that the names of the external entities, data stores and other processes which act as sources and sinks have been added. This completes the DFD hierarchy. A DFD hierarchy may have more levels in the cases where the specification has more layers of detail. In such cases, more detailed diagrams should be produced until no further decomposition of any operation into parts is useful.

Data flow diagrams are very useful for describing the system, but only when the specification has been written in the appropriate "state plus operations" style.

4. Other Diagrams

This section proposes some other diagrams which may be useful and help to explain a Z specification.

4.1 Showing the Detail of Schema Composition

The first sort of diagram in this section is to show the detail of schema composition. The purpose of schema composition is to take the state-after components of one schema and join them to the state-before components of the second. A diagram can be used to show which components of the schemas involved are affected. For example, the following definition in the example specification:

```
StackShelves ≙ StackFictionShelves⨟
              StackNonfictionShelves⨟StackReferenceShelves
```

will give rise to the diagram shown in Figure 17 below.

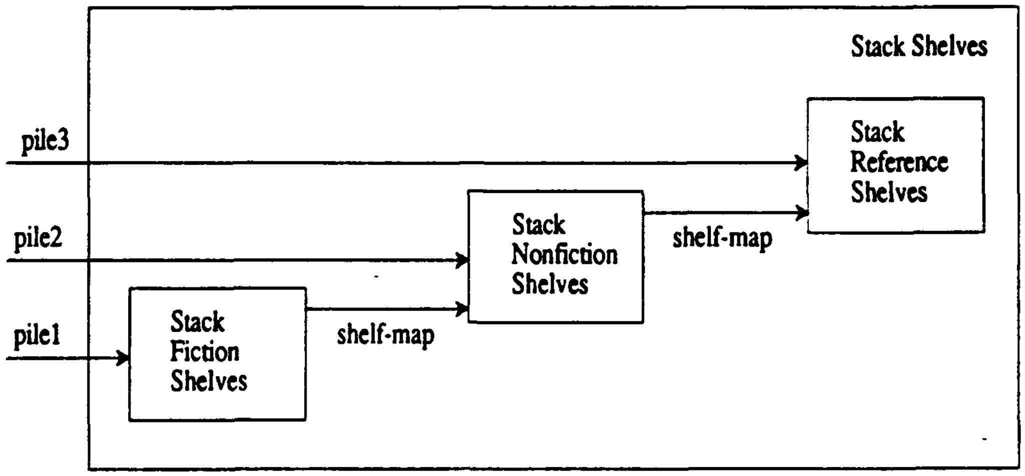

Figure 17 - Showing the Detail of Schema Composition

In this diagram, arrows indicate the direction of data flow to and from the operations. These are labelled with the base name of the component (without decorations or primes). Thus it may be seen that the first part of the operation, "Stack Fiction Shelves", takes "pile1?" as input and affects the state of the "shelf-map". The changed "shelf-map" is passed to the second part, "Stack Nonfiction Shelves", which takes "pile2?" as input and itself changes the "shelf-map". This is then passed to the final part, "Stack Reference Shelves", which takes "pile3?" as input. No output is produced.

This diagram gives a way to check at a glance what is happening when schemas are composed. That is, it is easy to see which components are involved.

4.2 Showing the Detail of Schema Piping

The second sort of diagram is to show the detail of schema piping. Piping is not part of Spivey's Z [1], but is supported by the ZADOK tool [8,9]. The purpose of schema piping is similar to schema composition, but rather than link state variables it takes the outputs of one schema as the inputs to the second. For example, the specification of the library system at Annex A contains the following definition:

```
ProcessNewBook ≙ ReceiveBook ≫ ClassifyBook
```

This operation may be described by the diagram in Figure 18 below.

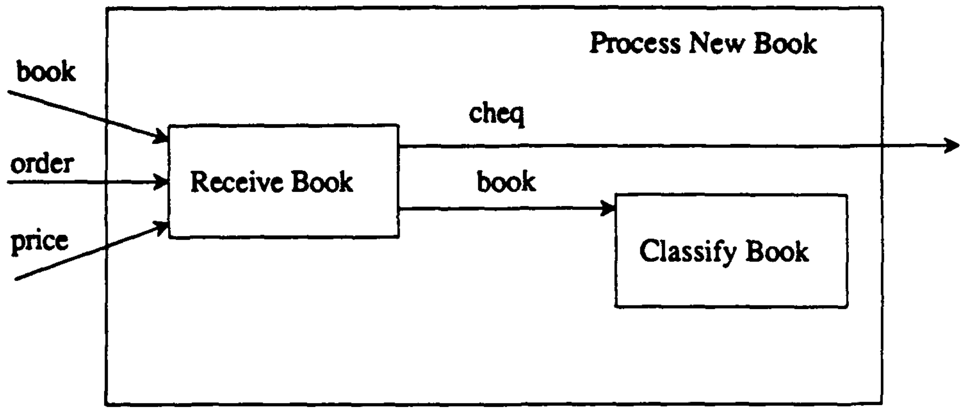

Figure 18 - Showing the Detail of Schema Piping

As for the schema composition diagram, arrows indicate the direction of data flow to and from the operations. These are labelled with the base name of the component (without the decorations "?" and "!"). Thus it may be seen that the "Receive Book" operation takes in three inputs, called "book?", "order?" and "price?", and produces two outputs, "cheq!" and "book!". Of these two outputs, one is piped into the second operation, to classify the book. The inputs and outputs of the complete process are those which cross the boundary of the outer box. With this sort of diagram, what happens to the inputs and outputs when two schemas are piped can easily be seen.

4.3 Showing Other Schema Operations

The map diagrams proposed in section 2.2 above show only the links between schemas, they give no detail about the sort of link. That is, is one schema included in another, or are two schemas conjoined to give a third? The basic map may be extended to make these distinctions. For example, consider Figure 19 below.

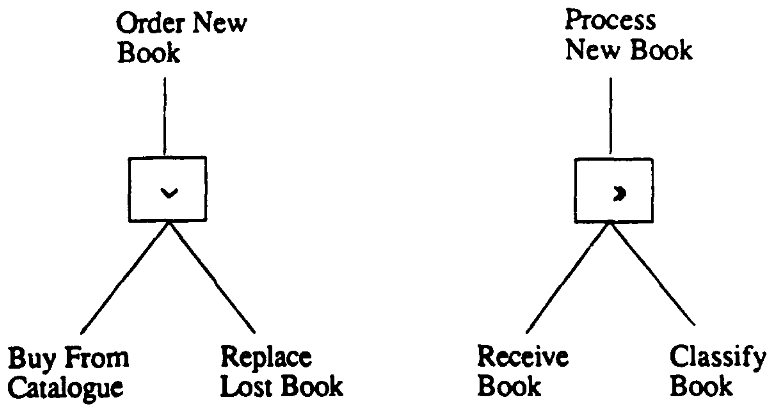

Figure 19 - Showing Schema Operators

The Z definitions corresponding to this diagram are:

$$\mathit{OrderNewBook} \mathrel{\widehat{=}} \mathit{BuyFromCatalogue} \lor \mathit{ReplaceLostBook}$$

and

$$\mathit{ProcessNewBook} \mathrel{\widehat{=}} \mathit{ReceiveBook} \gg \mathit{ClassifyBook}$$

The order of the names across the page is important for the definition of the "Process New Book" operation, as the order of piping is important. That is, schema1 $\gg$ schema2 is not the same as schema2 $\gg$ schema1. This is not the case for the "Order New Book" process, as disjunction is commutative. Another sort of diagram which says more about two schemas piped was proposed in section 4.2 above.

4.4 Structure Charts

Rather than drawing simple trees to describe data as proposed in section 2.3 above, more informative diagrams can be drawn, showing how data is constructed from basic elements. The diagrams are based on Jackson diagrams [10,11], and are also linked to the data dictionary notation described in [12].

All types in Z are built up from either given sets or simple branches of free types (constants) or both. In the diagrams proposed in this section, these basic elements are the leaves of a tree-shaped diagram, as they were in the maps proposed in section 2.3 above. However, additional information is added to describe how the data is constructed, that is, whether a schema has been used, or a free type, or a sequence.

The diagrams are best illustrated by example.

First, consider the following Z schema:

```
┌ loan ──────┐
│ n : name
│ due : date
└────────────┘
```

The structure chart corresponding to this schema is shown in Figure 20 below.

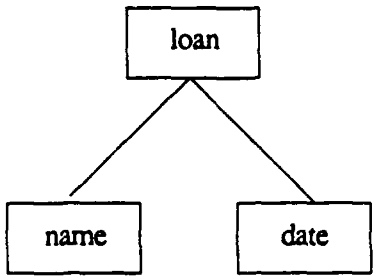

Figure 20 - The Structure Chart for the "loan" Schema

This chart says that a loan is constructed from a name and a date, in any order. The leaves of the diagram, "name" and "date", are given sets, that is, they cannot be further decomposed.

Now consider the following Z definition:

```
loan_record == seq loan
```

The structure chart corresponding to this definition is shown in Figure 21 below.

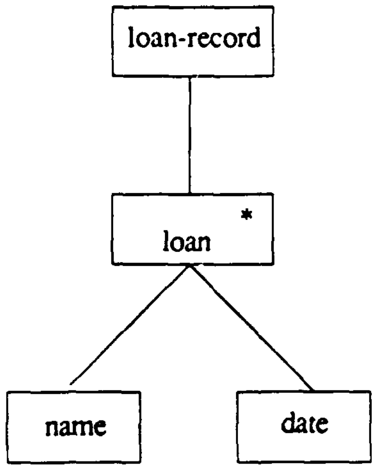

Figure 21 - The Structure Chart for the "loan-record" Sequence

This chart is again a tree shape, with given sets as leaves. The diagram has been annotated by the addition of an asterisk in the "loan" box. This shows that a loan-record is an iteration (sequence) of loans. Each loan is constructed from a name and a date as before.

The other sort of annotation on a structure chart is a small circle, which is used to indicate a selection. To illustrate this, consider the following free type definition:

```
book_type ::= fiction | nonfiction | reference
```

The structure chart corresponding to this is shown in Figure 22 below.

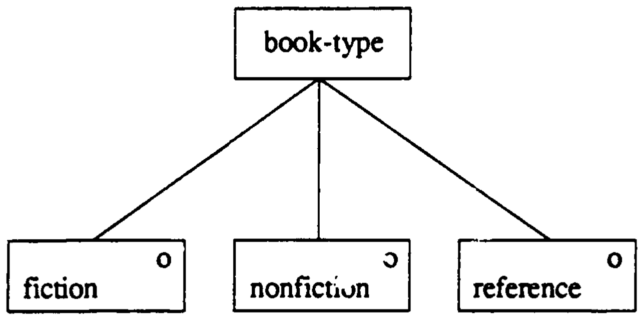

Figure 22 - The Structure Chart for the "book-type" Free Type

Note that in this diagram there are three leaves, corresponding to the three constants "fiction", "nonfiction" and "reference". A small circle is drawn in all three boxes to indicate a selection, that is, an element of the type "book-type" may be only one of "fiction", "nonfiction" and "reference".

Using these annotations, complicated structure charts can be built up. As an example, consider the following Z schema:

```
┌ library_book ────────┐
│ b : book
│ b_id : book_id
│ b_ty : book_type
│ l_rec : loan_record
└──────────────────────┘
```

The types in this schema are the given sets "book" and "book-id", the free type "book-type" and the sequence "loan-record". The corresponding structure chart is shown in Figure 23 below.

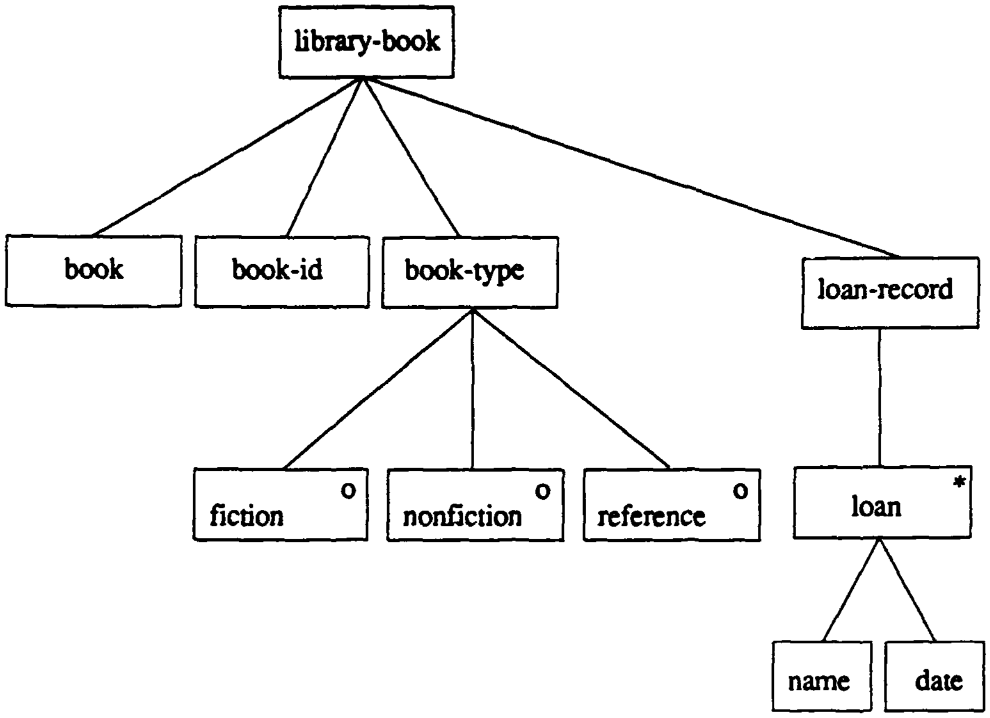

Figure 23 - The Structure Chart for the "library-book" Schema

As before, the leaves are the given sets and the constants from the free type definition. So, a library book is constructed from: a book and a book-id, which are both given sets; a book-type, which is one of "fiction", "nonfiction" and "reference"; and a loan record, which is a sequence of loans, each of which is constructed from a name and a date.

4.5 Entity Life Histories

Entity life histories are used to show what can happen to an entity, and in what order. These diagrams are again based on Jackson diagrams, and are also used in SSADM [13]. These diagrams cannot be generated automatically from a Z specification - historical information is not usually part of a Z specification and nor is time-ordering of operations. However, the operations that affect an entity, and appear on the entity life history, are operations which are defined in the Z specification. Figure 24 below shows an entity life history for a book from the library system example at Annex A.

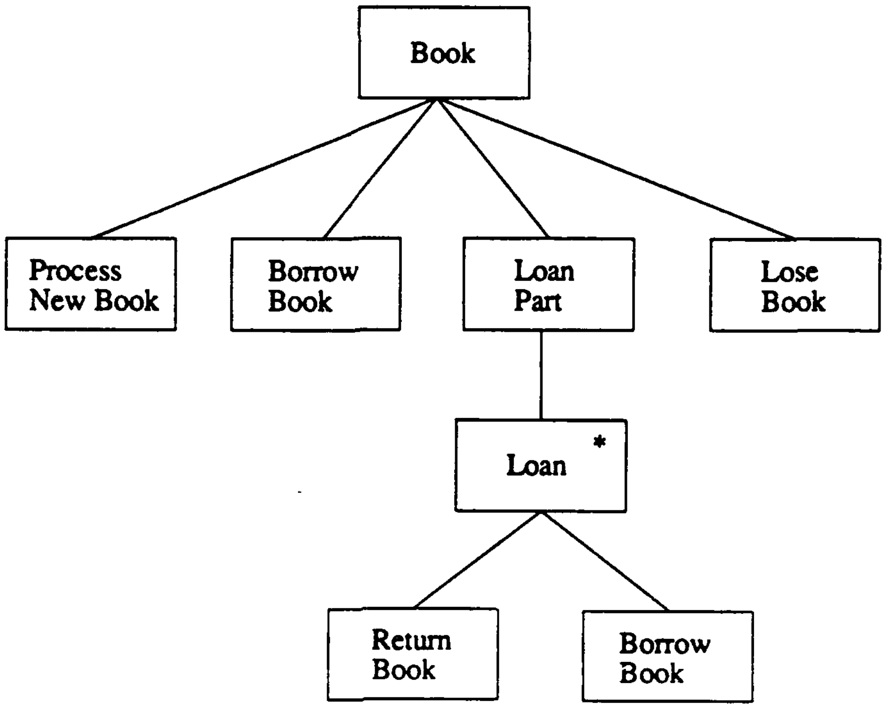

Figure 24 - An Entity Life History for a Book

A book starts life in the system when it arrives from the book shop, and ends when it is lost. In between these events, it is first borrowed, and then repeatedly returned and borrowed again until it is lost. The operations which affect the book are all specified as operation schemas in the formal specification.

The advantage of these diagrams is that they show how the operations defined in the Z specification are used. That is, they provide a framework for adding more information about the way the system is to work.

4.6 Venn Diagrams

Venn diagrams are a well-known way of illustrating relationships between sets, and, as Z is based on set theory, may well be useful to illustrate a Z specification. Examples of Venn diagrams are given in Figure 25 below.

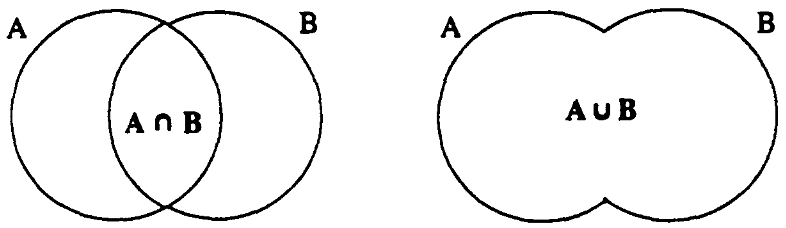

Figure 25 - Venn Diagrams

The diagram on the left shows the intersection (the shaded region), of two sets A and B. The sets may be schemas, or ordinary sets perhaps described by predicates. In the former case this diagram shows the conjunction of the schemas, and in the latter case it shows the intersection of the sets. Similarly, the diagram on the right shows the union of A and B (the shaded region). As before, A and B may be schemas or sets. If the former then this diagram shows the disjunction of the schemas and if the latter then it shows set union.

5. Conclusions

This paper has given examples of many diagrams which could be used to illustrate a Z specification, at various levels of detail. It is not suggested that all these diagrams should be produced for all specifications, just those which are believed to be useful in any particular case.

The diagrams proposed fall into three categories: maps, data flow diagrams, and others. Maps are used to show the structure of a specification, from the module level down to the structure of data. Also, maps can show how the schemas in a specification are related. Data flow diagrams, on the other hand, illustrate the movement of data around the system being specified and which operations affect which parts of the state. Such diagrams are only applicable to a specification written in a "state plus operations" style. The final category contains diagrams which show the detail of schema composition and piping as well as other schema operations such as disjunction or conjunction. These diagrams expand the maps by describing how schemas are built up by operations, rather than just schema inclusions. Other diagrams proposed are: structure charts, which are an alternative to the maps proposed in section 2.3; entity life histories, to show the order in which operations can affect things in the system; and Venn diagrams, to show the relationships between sets.

Diagrams make Z specifications much easier to understand, thus making them easier to validate. Diagrams help the writer of a specification, as they show the overall structure of the specification and how different parts of it combine. This high level view of a Z specification is necessary for understanding, and is very difficult to obtain from reading the formal text itself. The bottom-up approach of Z makes this particularly evident.

If diagrams are to be used with any certainty that they accurately reflect the current version of a specification, then they must be able to be generated automatically, or at least in a tool-supported way. With the maps discussed in section 2, this should not be a problem. All the information needed is in the Z specification - it should require little effort to modify an existing Z syntax- and type-checking tool to produce these diagrams. Of course, the layout of the diagrams will need to be human-assisted to make them easy to read. Similarly, rules have been formally specified to produce data flow diagrams from a Z specification. Thus tool support for this process is possible. Some of the diagrams described in section 4 pose more problems, particularly entity life histories, as the information required to produce them is not usually part of a Z specification.

In any case, a relationship must be defined between the diagrams and the formal specification to check that they are compatible. This relationship is vital for ensuring compatibility when the specification is updated or changed in any way.

This memorandum is the first step towards determining which diagrams should be produced, by proposing a set of options to be considered. Comments on the proposed diagrams are sought from the wider Z community. It is hoped that mechanisms for producing diagrams will be incorporated into future Z support tools.

References

[1] J M Spivey, "The Z Notation: A Reference Manual", Prentice-Hall International, 1989

[2] I Hayes (editor), "Specification Case Studies", Prentice-Hall International, 1987

[3] G P Randell, "Translating Data Flow Diagrams into Z (and vice versa)", RSRE Report 90019, October 1990

[4] G P Randell, "Improving the Translation from Data Flow Diagrams into Z by Incorporating the Data Dictionary", RSRE Report 92004, January 1992

[5] R Macdonald, "Z Usage and Abusage", RSRE Report 91003, February 1991

[6] "The Yourdon Structured Method (YSM) - An Introduction", Yourdon Inc.

[7] A Smith, "On Recursive Free Types in Z", RSRE Report 91028, August 1991

[8] C T Sennett, "Review of Type Checking and Scope Rules of the Specification Language Z", RSRE Report 87017, 1987

[9] G P Randell, "ZADOK User Guide", RSRE Memorandum 4356, 1990

[10] M Jackson, "System Development", Prentice-Hall International, 1983

[11] J R Cameron, "An Overview of JSD", IEEE Transactions on Software Engineering, vol SE-12 no 2, February 1986

[12] P T Ward & S J Mellor, "Structured Development for Real-Time Systems Volume 1: Introduction & Tools", Yourdon Press, 1985

[13] G Longworth & D Nicholls, "SSADM Manual Version 3", NCC Publications, December 1986

Acknowledgement

The author would like to thank Clare Harrold, Andrew Wood and Ruaridh Macdonald (all of DRA, Malvern) for their suggestions and comments on the sorts of diagrams they would like to see illustrating a Z specification.

Annex A - An Example Z Specification

A1. Introduction

This annex contains a Z specification of a library system. The specification is in five parts:

1. The specification of the types of data which are used by the different sub-systems of the library.
2. The book-buying sub-system which is responsible for ordering new books from a bookshop, paying for them when they arrive, and preparing them for being put on the library shelves ready to be loaned to borrowers.
3. The borrowers sub-system which is responsible for keeping track of registered borrowers and the books they have out on loan, and includes functions to add new borrowers, for a borrower to take out and return a book, and to lose a book.
4. The shelf-stackers sub-system, which is responsible for stacking returned and new books on the shelves, and for sending damaged books away to the bookbinders for repair and getting them back again.
5. The overall system, where the operations which cross the sub-system boundaries are specified. This specification actually only contains one such operation.

A2. Basic Types

This section introduces the basic types which are needed for the rest of the specification.

```
[ book ]
```

The library is concerned with books.

```
[ author, publication_date, publisher, title ]
```

There are various pieces of information associated with a book. These are its author, the date it was published, the publisher and its title.

```
┌ book_info ─────────────┐
│ ti : title
│ au : author
│ pub : publisher
│ p_d : publication_date
└────────────────────────┘
```

For convenience, all these pieces of information are grouped together into one schema.

```
book_type ::= fiction | nonfiction | reference
```

Books also have types, that is, they may be a fiction book, a non-fiction book or a reference book. It is possible to have two copies of the same book classified differently. For example, one copy of a dictionary may be classed as a non-fiction book and available for loan while another copy may be classed as a reference book and kept permanently in the library.

```
┌ book_data ──────┐
│ b_i : book_info
│ b_t : book_type
└─────────────────┘
```

The information about a book and its type are grouped together into one schema.

```
[ book_id ]
```

The library system has to be able to distinguish between two copies of the same book, so a set of unique identifiers for books is introduced.

Books which have been bought are not put directly onto the shelves of the library. They are first prepared by being stamped with their identifier and type, and have a loan record sheet stuck in them to record the borrowers and the date due back when they are loaned out.

```
[ date, name ]
```

The library has access to a calender, which is used to look up the date a book will be due back before it is borrowed. The set of borrowers' names is also introduced.

```
┌ 'oan ──────────┐
│ n : name
│ due : date
└────────────────┘
```

```
loan_record == seq loan
```

The loan record sheet is a list of all the borrowers who have borrowed that book and the date by which they should have returned it.

```
┌ library_book ──────────┐
│ b : book
│ b_id : book_id
│ b_ty : book_type
│ l_rec : loan_record
└────────────────────────┘
```

A book which has been prepared is stamped with its identifier and type, and has a loan record sheet (initially empty) stuck in it. It is now a library book.

This completes the specification of the basic types. These types will b used in the subsequent four sections, which describe the three subsystems which make up the library system, and the overall system itself.

A3. The Book-Buying Sub-System

This section specifies one of the sub-systems, namely the system responsible for buying new books. It uses information from the previous section on types.

```
┌ BooksDatabase ─────────────────────┐
│ bookmap : book_id ⇸ book_data
└────────────────────────────────────┘
```

All of the data pertaining to books is held in one database. This database is indexed by the unique book identifier.

```
┌ Trolley ──────────────┐
│ pile : ℙ library_book
└───────────────────────┘
```

The pile of books ready to go onto the shelves are put on a trolley. The actual task of putting the books out is done by another department of the library (specified in Section A5).

Books are ordered for two reasons. The first is that a book is lost by a borrower, and the second is that a new book is seen in a catalogue from a bookshop.

```
[ price ]
```

Books cost money, so each book has a price.

```
┌ catalogue_entry ──┐
│ b_i  : book_info
│ pr   : price
└───────────────────┘
```

```
catalogue == seq catalogue_entry
```

The catalogue is just a list of entries, represented in Z as a sequence. Each entry consists of the description of the book (the book information) and its price.

In order to buy a book, an order must first be placed.

```
[ address, order_number ]
```

Some more sets are needed to represent addresses so the book shop knows where to send the book, and order numbers to allow each separate order form to be easily identified.

```
│ our_address : address
```

A particular address is introduced.

```
┌ order_form ─────────┐
│ o_n  : order_number
│ b_i  : book_info
│ add  : address
└─────────────────────┘
```

An order form has a number, the information describing the book wanted, and the address to send the book to. Only one book is ordered per form.

```
┌ OrdersPending ──────────────────────────────┐
│ ordermap : order_number ⇸ order_form
├─────────────────────────────────────────────
│ ∀on:dom ordermap · (ordermap on).o_n = on
└─────────────────────────────────────────────┘
```

The library keeps track of all orders which have been sent but for which no book has yet been received. The order number used to index each order form must be one which appears on the order form itself.

```
| right_info : book ↔ book_info
```

The information for a given book needs to be checked to ensure it is right. That is, whether the title recorded for it is the one which appears on its front cover, etc. The exact definition of this relation is left unspecified.

```
| right_type : book ↔ book_type
```

A check is also needed to ensure that a book has been given the right type. That is, that a novel is not classified as a non-fiction book. The exact definition of this relation is again left unspecified.

The operations performed by this sub-system, those of ordering a book, receiving a book, and classifying a book may now be defined.

The first operation is to order a new book from the catalogue.

```
┌ BuyFromCatalogue ──────────────────────────
│ ΞBooksDatabase
│ ΔOrdersPending
│ cat? : catalogue
│ order! : order_form
├─────────────────────────────────────────────
│ order!.o_n = new_number
│ order!.b_i = this_book
│ order!.add = our_address
│ ordermap' = ordermap ∪ {new_number ↦ order!}
│ where
│  | new_number : order_number
│  | this_book : book_info
│  | c_e : catalogue_entry
│  ├──────────────────────────────────────────
│  | c_e ∈ rng cat? ∧ ce.b_i = this_book
│  | new_number ∉ dom order_map
└─────────────────────────────────────────────
```

This operation specifies that the order form that sent out is correctly filled in. That is, it must be given a new number which has not already been used (to allow the order to be easily identified); the book requested must be from the catalogue; and the address must be correctly filled in to ensure that the book is sent to the requesting library and not to somewhere else. A copy of this order form is then added to the pending orders file.

Books are also bought to replace lost ones.

```
┌ ReplaceLostBook ──────────────────────────────────────
│ ΔBooksDatabase
│ ΔOrdersPending
│ lost? : book_id
│ order! : order_form
├───────────────────────────────────────────────────────
│ order!.o_n = new_number
│ order!.b_i = (bookmap lost?).b_i
│ order!.add = our_address
│ ordermap' = ordermap ∪ {new_number ↦ order!}
│ bookmap' = { lost? } ⩤ bookmap
│ where
│  │ new_number : order_number
│  ├──────────────────────────────
│  │ new_number ∉ dom order_map
└───────────────────────────────────────────────────────
```

Replacing a lost book is similar to buying a book from a catalogue, in that an order is sent to the bookshop and a copy put on file. The books database is consulted to find the book information needed, using the identifier it receives from the borrowers sub-system. The record of the lost book is removed from the books database to allow the identifier to be re-used when the new copy comes in (if required).

```
OrderNewBook ≙ BuyFromCatalogue ∨ ReplaceLostBook
```

Every book ordered is either bought from a catalogue, or is bought as a replacement for a lost book.

The second operation is the one which receives the book from the book shop.

```
[ cheque ]
```

The set of cheques is introduced, to allow the library to pay for books it has received.

```
│ right_amount : cheque ↔ price
```

A cheque must be for the right amount. The exact form of this relation is left unspecified.

```
┌ ReceiveBook ──────────────────────────────────────
│ ΔOrdersPending
│ book?, book! : book
│ order? : order_form
│ price? : price
│ cheq! : cheque
├───────────────────────────────────────────────────
│ order? ∈ rng ordermap ∧ order?.add = our_address
│ right_info (book?, order?.b_i)
│ ordermap' = ordermap ⩥ { order? }
│ right_amount (cheq!, price?)
│ book! = book?
└───────────────────────────────────────────────────
```

This process checks that: 1) the order received is one that has actually been sent to the book shop; 2) the book is the right one, that is, that its title, author etc. match the information on the order form; and 3) the order has the right address on it. (The third predicate is actually unnecessary, as it is implied by the first. But it helps to make things clear.) If these checks are passed then the copy of the order on the pending file is removed. A cheque is sent back to the bookshop for the right amount to pay for the book, and the book is passed on to the next stage.

```
┌ ClassifyBook ─────────────────────────────────────
│ ΔBooksDatabase
│ ΔTrolley
│ book? : book
├───────────────────────────────────────────────────
│ pile' = pile ∪ { lib }
│ bookmap' = bookmap ∪ { bid ↦ bd }
│ where
│   │ bid : book_id
│   │ bd : book_data
│   │ lib : library_book
│   ├───────────────────────────────────────────────
│   │ bid ∉ dom bookmap
│   │ right_info (book?,bd.b_i) ∧ right_type (book?,bd.b_t)
│   │ lib.b = book? ∧ lib.b_id = bid
│   │ lib.b_ty = bd.b_t ∧ lib.l_rec = ⟨⟩
└───────────────────────────────────────────────────
```

The process of classifying a book involves preparing the book by stamping it with its identifier and classification and sticking an empty loan record sheet in it, and then adding it to the pile on the trolley. The information about the book is recorded in the books database.

```
ProcessNewBook ≙ ReceiveBook ≫ ClassifyBook
```

A new book is processed in these two stages.

This completes the specification of the book-buying sub-system.

A4. The Borrowers Sub-System

```
[ borrower_id ]
```

A set of identifiers is introduced, so that the library can assign a unique identifier to each borrower to allow them to be distinguished.

```
┌ application_form ──┐
│ n : name
│ ad : address
└────────────────────┘
```

When a person wishes to join the library and become a borrower, they submit an application form. This form has their name and address on it.

```
valid_forms : ℙ application_form
```

The forms that are correctly filled in are valid.

```
┌ membership_card ──────────┐
│ b_id : borrower_id
└───────────────────────────┘
```

The library issues a membership card to each borrower with the identifier assigned to that borrower on it.

```
┌ Borrowers ──────────────────────────────────────────┐
│ borrowers : borrower_id ⤀ application_form
└─────────────────────────────────────────────────────┘
```

The library maintains a record of its borrowers, by keeping the application form for each in a file. The unique identifier assigned to each borrower is used as the index for the file.

```
┌ AddNewBorrower ─────────────────────────────
│ ΔBorrowers
│ ap? : application_form
│ mc! : membership_card
├─────────────────────────────────────────────
│ ap? ∈ valid_forms
│ mc!.b_id = bi
│ borrowers' = borrowers ∪ { bi ↦ ap? }
│ where
│  │ bi : borrower_id
│  ├──────────────────────────────────────────
│  │ bi ∉ dom borrowers
└─────────────────────────────────────────────
```

When a person wishes to become a borrower, they submit an application form. If the form is valid then a new borrower identifier is assigned, and the form filed. The new borrower is given a membership card with the borrower identifier on it.

```
┌ Books_on_loan ──────────────────────
│ on_loan : book_id ⇸ borrower_id
└─────────────────────────────────────
```

The library has a file in which to record who has which books out on loan at the current time. Any one borrower may have several books out at once, but one book can only be with one borrower at a time.

```
┌ BorrowBook ─────────────────────────────────────────┐
│ ΞBorrowers
│ ΔBooks_on_loan
│ due? : date
│ mc?, mc! : membership_card
│ lb?, lb! : library_book
├─────────────────────────────────────────────────────
│ mc?.b_id ∈ dom borrowers
│ lb?.b_ty ≠ reference
│ on_loan' = on_loan ∪ { lb?.b_id ↦ mc?.b_id }
│ lb!.b = lb?.b ∧ lb!.b_id = lb?.b_id
│ lb!.b_ty = lb?.b_ty ∧ lb!.l_rec = lb?.l_rec ⁀ <l>
│ mc! = mc?
│ where
│  │ l : loan
│  ├──────────────────────────────────────────────────
│  │ l.n = ( borrowers (mc?.b_id) ).n ∧ l.due = due?
└─────────────────────────────────────────────────────┘
```

To borrow a book, a borrower gives the book to the librarian along with a membership card. The librarian checks that the borrower's identifier is a valid one, and that the book is not a reference book. The loan record sheet in the book is updated with the borrower's name and the date by which it must be returned. This date is found from the calendar. The file of books on loan is updated to show this loan and the book and membership card are then returned to the borrower.

```
┌ ReturnBook ──────────────────────┐
│ ΔBooks_on_loan
│ ΔTrolley
│ lb? : library_book
├──────────────────────────────────
│ lb?.b_id ∈ dom on_loan
│ on_loan' = { lb?.b_id } ⩤ on_loan
│ pile' = pile ∪ { lb? }
└──────────────────────────────────┘
```

When a book is returned, the librarian checks that it was one which had been recorded as being out on loan. The file of books on loan is then updated by removing the record for this book, and the book is put on the trolley. The task of putting the book back is carried out by another department of the library, specified in section A5.

```
┌ LoseBook ──────────────────────────
│ ΔBooks_on_loan
│ mc?, mc! : membership_card
│ lost?, lost! : book_id
├────────────────────────────────────
│ lost? ↦ mc?.b_id ∈ on_loan
│ on_loan' = { lost? } ⩤ on_loan
│ mc! = mc? ∧ lost! = lost?
└────────────────────────────────────
```

Sometimes a borrower loses a book. When this happens, the borrower presents a membership card to the librarian, and says which book has been lost. A check is made that the borrower had that book on loan, and the record of this loan is removed from the on loan list. The borrowers card is returned, and the identifier of the book concerned is passed to the book buying sub-system to replace it. A more realistic library would probably impose a fine for losing a book!

This completes the specification of the borrowers sub-system.

A5. The Shelf-Stackers Sub-System

```
[ location ]
```

Every book has a location, that is, where it is on the shelf in the library.

```
fiction_books == {library_book | b_ty = fiction}
nonfiction_books == {library_book | b_ty = nonfiction}
reference_books == {library_book | b_ty = reference}
```

Library books are divided into three sets, depending on their type (fiction, non-fiction or reference).

This sub-system is concerned with the shelves of the library.

```
┌ Shelves ───────────────────────────
│ shelf_map : library_book ⇸ location
└────────────────────────────────────
```

The library shelves consist of library books at their locations.

```
┌ SortBooks ─────────────────────────────
│ ΔTrolley
│ pile1! : ℙ fiction_books
│ pile2! : ℙ nonfiction_books
│ pile3! : ℙ reference_books
├────────────────────────────────────────
│ ( pile1! ∪ pile2! ∪ pile3! ) = pile
│ pile' = {}
└────────────────────────────────────────
```

The operation to sort the books from the trolley ready to put them back on the shelves takes all the books and puts them into three piles, one for each type. The trolley ends up empty.

```
│ is_ordered : ℙ (library_book ⇸ location)
```

A shelf must be ordered correctly. The actual definition of being ordered is left unspecified.

There are three parts to stacking the shelves, one for each of the type of books being stacked. When a pile of books, of whichever type, is stacked on a shelf, the books are put in the right places so that the shelf ends up properly ordered.

```
┌ StackFictionShelves ──────────────────
│ ΔShelves
│ pile1? : ℙ fiction_books
├────────────────────────────────────────
│ dom shelf_map' = dom shelf_map ∪ pile1?
│ is_ordered (shelf_map')
└────────────────────────────────────────
```

```
┌ StackNonfictionShelves ───────────────
│ ΔShelves
│ pile2? : ℙ nonfiction_books
├────────────────────────────────────────
│ dom shelf_map' = dom shelf_map ∪ pile2?
│ is_ordered (shelf_map')
└────────────────────────────────────────
```

```
┌ StackReferenceShelves ──────────────
│ ΔShelves
│ pile3? : ℙ reference_books
├────────────────────────────────────
│ dom shelf_map' = dom shelf_map ∪ pile3?
│ is_ordered (shelf_map')
└────────────────────────────────────
```

```
StackShelves ≙ StackFictionShelves⨟
               StackNonfictionShelves⨟StackReferenceShelves
```

The shelves are stacked by stacking each type of book in turn.

```
┌ SendBookforRepair ──────────────
│ ΔShelves
│ book! : library_book
├────────────────────────────────
│ book! ∈ dom shelf_map
│ shelf_map' = {book!} ⩤ shelf_map
└────────────────────────────────
```

A book which is *currently on the shelf* can be removed and sent for repair when it is damaged.

```
┌ ReceiveMendedBook ───────
│ ΔTrolley
│ book? : library_book
├─────────────────────────
│ pile' = pile ∪ {book?}
└─────────────────────────
```

When a book is mended, it is sent back to the library and is put on the trolley, from where it will be stacked back on the shelves.

This completes the Z specification of the shelf-stackers sub-system.

A6. The Overall System

The final part of this specification specifies those operations which cross sub-system boundaries. In this example, there is only one, and that is concerned with replacing a lost book. The operation is specified as follows:

$\text{LostBook} \mathrel{\widehat{=}} \text{LoseBook} \gg \text{ReplaceLostBook}$

The identifier of the lost book from the borrowers department is sent to the book buying department who order a new book to replace it.

This completes the example specification.

REPORT DOCUMENTATION PAGE DRIC Reference Number (if known) ....

Overall security classification of sheet ..........UNCLASSIFIED..........
(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification, eg (R), (C) or (S).

Originators Reference/Report No. MEMO 4587	Month MARCH	Year 1992
Originators Name and Location DRA, ST ANDREWS ROAD MALVERN, WORCS WR14 3PS		
Monitoring Agency Name and Location		
Title Z THROUGH PICTURES		
Report Security Classification UNCLASSIFIED	Title Classification (U, R, C or S) U	
Foreign Language Title (in the case of translations)		
Conference Details		
Agency Reference	Contract Number and Period	
Project Number	Other References	
Authors RANDELL, G P	Pagination and Ref 42	

Abstract

The purpose of this memorandum is to propose diagrams which may be generated from a Z specification either fully automatically or with human intervention. The main purpose of such diagrams is to illustrate the specification and thus help a reader to understand it and to validate it. Examples of the diagrams proposed are given.

Abstract Classification (U, R, C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

580/48